An Evaluation of the TRIPS Computer System (Extended Technical Report)[1]

Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz
Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill
Stephen W. Keckler, Doug Burger, Kathryn S. McKinley

Computer Architecture and Technology Laboratory
Department of Computer Sciences
Technical Report TR-08-31
The University of Texas at Austin
cart@cs.utexas.edu — www.cs.utexas.edu/users/cart

Abstract

The TRIPS system employs a new instruction set architecture (ISA) called Explicit Data Graph Execution (EDGE) that renegotiates the boundary between hardware and software to expose and exploit concurrency. EDGE ISAs use a block-atomic execution model in which blocks are composed of dataflow instructions. The goal of the TRIPS design is to mine concurrency for high performance while tolerating emerging technology scaling challenges, such as increasing wire delays and power consumption. This paper evaluates how well TRIPS meets this goal through a detailed ISA and performance analysis. We compare performance, using cycles counts, to commercial processors. On SPEC CPU2000, the Intel Core 2 outperforms compiled TRIPS code in most cases, although TRIPS matches a Pentium 4. On simple benchmarks, compiled TRIPS code outperforms the Core 2 by 10% and hand-optimized TRIPS code outperforms it by factor of 3. Compared to conventional ISAs, the block-atomic model provides a larger instruction window, increases concurrency at a cost of more instructions executed, and replaces register and memory accesses with more efficient direct instruction-to-instruction communication. Our analysis suggests ISA, microarchitecture, and compiler enhancements for addressing weaknesses in TRIPS and indicates that EDGE architectures have the potential to exploit greater concurrency in future technologies.

[1] **This technical report contains per benchmark data for the SPEC benchmarks in the ISA section, per benchmark memory footprint data in the ISA section, detailed characterization data of the memory system and operand network in the microarchitecture section, and full performance data across all reference platforms in the comparison section that was not included due to space constraints in "An Evaluation of the TRIPS Computer System"; Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, Kathryn S. McKinley; ASPLOS 2009, Washington DC, USA; March 2009.**

An Evaluation of the TRIPS Computer System (Extended Technical Report)

Abstract

The TRIPS system employs a new instruction set architecture (ISA) called Explicit Data Graph Execution (EDGE) that renegotiates the boundary between hardware and software to expose and exploit concurrency. EDGE ISAs use a block-atomic execution model in which blocks are composed of dataflow instructions. The goal of the TRIPS design is to mine concurrency for high performance while tolerating emerging technology scaling challenges, such as increasing wire delays and power consumption. This paper evaluates how well TRIPS meets this goal through a detailed ISA and performance analysis. We compare performance, using cycles counts, to commercial processors. On SPEC CPU2000, the Intel Core 2 outperforms compiled TRIPS code in most cases, although TRIPS matches a Pentium 4. On simple benchmarks, compiled TRIPS code outperforms the Core 2 by 10% and hand-optimized TRIPS code outperforms it by factor of 3. Compared to conventional ISAs, the block-atomic model provides a larger instruction window, increases concurrency at a cost of more instructions executed, and replaces register and memory accesses with more efficient direct instruction-to-instruction communication. Our analysis suggests ISA, microarchitecture, and compiler enhancements for addressing weaknesses in TRIPS and indicates that EDGE architectures have the potential to exploit greater concurrency in future technologies.

1 Introduction

Growing on-chip wire delays, coupled with complexity and power limitations, have placed severe constraints on the issue-width scaling of conventional superscalar architectures. Because of these trends, major microprocessor vendors have abandoned architectures for single-thread performance and turned to the promise of multiple cores per chip. While many applications can exploit multicore systems, this approach places substantial burdens on programmers to parallelize their codes. Despite these trends, Amdahl's law dictates that single-thread performance will remain key to the future success of computer systems [8].

In response to semiconductor scaling trends, we designed a new architecture and microarchitecture intended to extend single-thread performance scaling beyond the capabilities of superscalar architectures. TRIPS is the first instantiation of these research efforts. TRIPS uses a new class of instruction set architectures (ISAs), called Explicit Data Graph Execution (EDGE), which renegotiate the hardware and software boundary. EDGE ISAs use a block-atomic execution model, in which EDGE blocks consist of dataflow instructions. This model preserves sequential memory semantics and exposes greater instruction level concurrency without requiring explicit software parallelization. We constructed a custom 170 million transistor ASIC, an instantiation of the ISA (TRIPS ISA), TRIPS system circuit boards, a runtime system, performance evaluation tools, and a compiler that optimizes and translates C and Fortran programs to the TRIPS ISA. The distributed processing cores of a TRIPS processor issue up to 16 instructions per cycle from an instruction window of up to 1024 instructions contained in 8 blocks. The TRIPS ISA and distributed microarchitecture are designed to exploit concurrency and reduce the influence of long wire delays by exposing the spatial nature of the microarchitecture to the compiler for optimization.

This paper presents a performance analysis that explores how well the TRIPS system meets its goals of exploiting concurrency, hiding latency, and distributing control. Using the TRIPS hardware and microarchitectural simulators, we use compiled and hand-optimized benchmarks to compare the EDGE ISA, microarchitecture, and performance to modern processors. While we measured the power consumed by one TRIPS processor and the memory system to be 17W (30W for the whole chip with two processors and no clock gating), a detailed power analysis and an examination of multicore execution is beyond the scope of this paper.

Our microarchitecture analysis shows that TRIPS can fill much of its instruction window; compiled code shows an average of 400 total instructions in flight (887 peak for the best benchmark) and hand-optimized code shows an average of 630 (1013 peak). While much higher than conventional processors, the number of instructions in flight is less than the maximum of 1024 because the compiler does not completely fill blocks and the hardware experiences pipeline stalls and flushes due to I-cache misses, branch mispredictions, and load dependence mispredictions. The EDGE ISA incurs substantial increases in instructions fetched and executed relative to conventional RISC architectures because of predication and instruction overheads required by the dataflow model. A strength of the EDGE ISA and distributed control is that TRIPS requires less than half as many register and memory accesses than a RISC ISA (Alpha in this paper) because it converts these into direct producer to consumer communications. Furthermore, communicating instructions are usually on the same tile or an adjacent tile, which makes them power efficient and minimizes latency.

We compare the performance (measured in cycles) of TRIPS to the Intel Core 2, Pentium III, and Pentium 4 using hardware performance counters on compiled and hand-optimized programs. On EEMBC, the Core 2 executes 30% fewer cycles than TRIPS compiled code. On SPEC2000, TRIPS compiled code executes more than twice as many cycles than Core 2 on integer benchmarks but the same number of cycles on floating-point benchmarks. TRIPS executes 3 times fewer cycles than the Core 2 on hand-optimized benchmarks. These experiments suggest that EDGE processors have the capability to achieve substantial performance improvements over conventional microprocessors by exploiting concurrency. However, realizing this potential relies on the compiler to better expose concurrency and create large blocks of TRIPS instructions, as well as microarchitectural innovations in control distribution and branch prediction.

2 The TRIPS Processor Architecture

The foundations of the TRIPS system were published in 2001 [15]. Between 2001 and 2004, we refined the architecture so as to realize it in silicon and began the compiler implementation. The TRIPS chip taped out in August 2006 and was fully functional (no known bugs) in the lab in February 2007. The TRIPS chip uses a 130nm ASIC technology and contains 170 million transistors. One chip contains two processors and the simplest system consists of four TRIPS chips. Each chip contains two processors and 2GB of local DRAM

RISC Code

```
Label1:
ld   %3, 4(%2)
blez %3, Label2
ld   %5, 8(%2)
addi %3, %3, %5
st   %3, 4(%2)
Label2:
addi %4, %4, #-1
bgez %4, Label1
Label3:
```

TRIPS EDGE Code

```
.bbegin block1
R0: read  $t2, $g2
R1: read  $t4, $g4
I0: mov    $t3, $t2
I1: ld     $t5, 4($t2)
I2: ld     $t6, 8($t3)
I3: tlez   $t7, $t5
I4: addi_f<$t7> $t8, $t5, $t6
I5: null_t<$t7> $t8
I6: st      $t8, 4($t3)
I7: subi    $t9, $t4, #1
I8: teqz    $t10, $t9
I9: b_t<$t10> block3
I10:b_f<$t10> block1
W0: write $g4, $t9
.bend block1
.bbegin block3 ...
```

Dataflow Graph

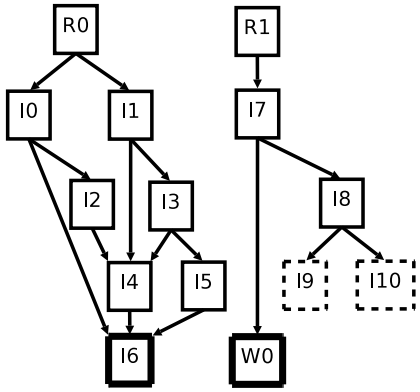

Figure 1: RISC and TRIPS code with dataflow graph.

connected to a motherboard. While we designed the system to scale to eight motherboards (64 processors), this paper examines a single TRIPS processor using single-threaded codes. We summarize the architecture and compiler below; details are in prior publications [2, 19, 22].

EDGE ISA: Two defining features of an Explicit Data Graph Execution (EDGE) ISA are *block-atomic* execution [13] and direct instruction communication within a block, which together enable efficient hybrid dataflow execution. An EDGE microarchitecture maps each compiler-generated dataflow graph to a distributed execution substrate. The ISA was designed to provide high-performance, single-threaded, concurrent, and distributed execution.

The TRIPS ISA aggregates up to 128 instructions in a block. The block-atomic execution model logically fetches, executes, and commits each block as a single entity. Blocks amortize per-instruction bookkeeping and reduce branch predictions, providing latency tolerance to make distributed execution practical. Blocks communicate through registers and memory. Within a block, *direct instruction communication* delivers results from producer to consumer instructions in dataflow fashion. This supports distributed execution by eliminating accesses to a shared register file.

Figure 1 compares RISC and TRIPS EDGE code on an example. The TRIPS register reads (R0, R1) at the beginning of the block start dataflow execution by injecting values from the register file into the block. The block ejects the register write (W0) and writes register `$g4` when the block commits. Instruction operands within the block, such as `$t2`, are passed directly from producer to consumer without an intervening register access. Because the instructions encode their targets, rather than a register in a common register file, a 32-bit instruction encoding has room for at most two targets. When more targets are required, such as the value

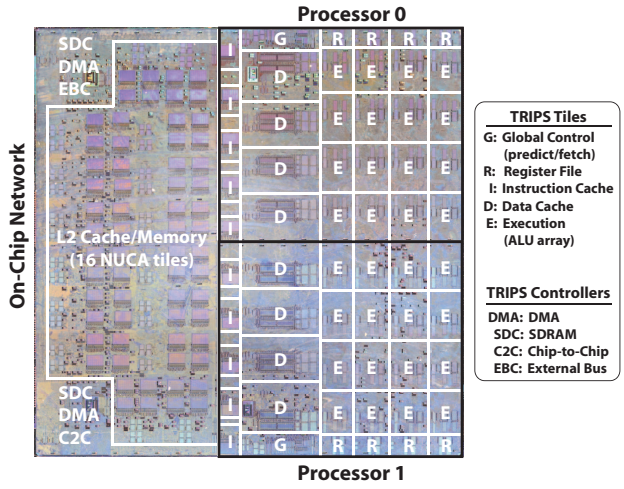

Figure 2: TRIPS die photo with tile overlays.

read in instruction R0, the program needs a `mov` (move) instruction (I0) to replicate the value flowing in the dataflow graph. The TRIPS code also shows that branch and non-branch instructions can be predicated. To enable the hardware to detect block completion, the execution model requires that all block outputs (register writes and stores) be produced regardless of the predicated path within the block. The `null` instruction produces a token that when passed through the `st` (store) indicates that the store output has been produced, but does not modify memory. In our experiments, we do not classify these dataflow execution helper instructions as useful when comparing to conventional ISAs. The dataflow graph shows the producer/consumer relationships encoded in the TRIPS binary.

TRIPS Microarchitecture: Because the goals of the TRIPS microarchitecture include scalability and distributed execution, it has no global wires, reuses a small set of components on routed networks, and can be extended to a wider-issue implementation without source recompilation or ISA changes. Figure 2 superimposes the tile-level block diagram on a TRIPS die photo. Each TRIPS chip contains two processors and a secondary memory system, each inter-connected by one or more micronetworks. Each processor uses five types of tiles: one global control tile (GT), 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT). The tiles communicate via six micronetworks that implement distributed control and data protocols. The main micronetwork is the operand network (OPN), which replaces a bypass network in a conventional superscalar. The two-dimensional, wormhole-routed, 5x5 mesh OPN delivers one 64-bit operand per link per cycle [7]. The other networks perform distributed instruction fetch, dispatch, I-cache refill, and completion/commit.

TRIPS fetches and executes each block *en masse*. The GT sends a block address to the ITs which deliver the block's computation instructions to the reservation stations in the 16 execution tiles (ETs), 8 per tile as

specified by the compiler. The ITs also deliver the register read/write instructions to reservation stations in the RTs. The RTs read values from the global register file and send them to the ETs, starting dataflow execution. The GT instigates the commit protocol once each DT and RT receives all block outputs. The commit protocol updates the data caches and register file with the speculative state of the block. The GT uses its next block predictor (branch predictor) to begin fetching and executing the next block while previous blocks are still executing. The prototype can simultaneously execute up to eight 128-instruction blocks (one non-speculative, seven speculative) giving it a maximum window size of 1024 instructions.

At 130 nm, each TRIPS processor occupies approximately 92 mm^2 of a total chip area of 330 mm^2. If scaled down to 65 nm, a TRIPS core would be approximately 23 mm^2, similar to the 29 mm^2 of a Core 2 processor. A direct comparison is difficult because TRIPS uses an ASIC technology and lacks some hardware needed for an operating system. Nonetheless, TRIPS has a greater density of arithmetic units in a similar area and the architecture provides greater issue width and instruction window scaling.

TRIPS Compiler: The TRIPS compiler first performs conventional optimizations such as inlining, unrolling, common subexpression elimination, scalar replacement, and TRIPS-specific optimizations such as tree-height reduction to expose parallelism. The compiler next translates the code to the TRIPS Intermediate Language (TIL), a RISC-like IR, and progressively transforms TIL into blocks that conform to the TRIPS block constraints: up to 128 instructions, up to 32 register read/writes with 8 per bank, and up to 32 load/store identifiers [22]. The compiler aggregates basic blocks from multiple control paths into optimized TRIPS blocks using predication, tail duplication, and loop optimizations [23, 12]. This process is similar to hyperblock formation, but more challenging because of the additional block constraints that simplify the hardware. The compiler iteratively merges and optimizes blocks until they are as full as possible and then performs register allocation. This phase produces completed TIL with correct and fully specified blocks, as in Figure 1.

The compiler's scheduler then transforms TIL to TRIPS assembly language (TASL), which includes a mapping of instructions to execution tiles. The scheduler seeks a mapping that exposes instruction concurrency and minimizes communication overheads (distance and contention) [3]. This mapping optimizes performance without restricting functional portability as the hardware can remap an EDGE binary to different hardware topologies (number of tiles) without recompilation or changes to the binary.

3 Evaluation Methodology

We evaluate the TRIPS system and compare its performance with conventional architectures using performance counters on the TRIPS hardware and on commercial platforms. Sections 4 and 5 present TRIPS and Alpha simulation results to gain insights into the relative strengths and weaknesses of TRIPS. All performance measurements in Section 6 are from actual hardware.

System	Issue Width	Proc Speed (MHz)	Mem Speed (MHz)	Proc/Mem Ratio	L1 Cap. (D/I) (KB)	L2 Cap. (MB)	Mem Cap. (GB)
TRIPS	16	366	200	1.83	32 / 80	1	2
Core 2	4	1600	800	2.00	32 / 32	2	2
Pentium 4	4	3600	533	6.75	16 / 150	2	2
Pentium III	3	450	100	4.50	16 / 16	0.5	0.256

Table 1: Reference platforms.

The TRIPS System: A TRIPS chip consists of two processors that share a 1 MB L2 static NUCA cache [10] and 2 GB of DDR Memory; we use one processor for all experiments in this study. Each processor has a private 32 KB L1 data cache and a private 80 KB L1 instruction cache. We run the processor core at 366 MHz and the DRAM with 100/200 MHz DDR clocks. TRIPS system calls interrupt program execution, halt the processor, and execute off-chip on a commercial processor running Linux. Because the TRIPS cycle counters increment only when the processor is not halted, the program performance measurements ignore the time to process system calls. The tools we use to measure cycles in the commercial systems also exclude operating system execution time, thus providing a fair comparison.

Simulators: We use functional and cycle-level TRIPS simulators to gather statistics not available from the hardware [25]. Validation of the TRIPS cycle counters against the TRIPS simulators indicates statistical differences of less than 5%. We use a customized version of the M5 simulator [1] to produces statistics that measure loads, stores, and register accesses from *gcc*-compiled Alpha-Linux binaries.

Reference Platforms: We compare TRIPS performance to three reference platforms from the Intel x86 product family (Pentium III, Pentium 4, and Core 2). Table 1 shows the platform configurations including processor and DDR DRAM clock speed and the memory hierarchy capacities. Because each machine is implemented in a different process technology, we compare cycle counts obtained from performance counters, using PAPI on the Intel processors [16]. Cycle count is an imperfect metric because some architectures, particularly the Pentium 4, emphasize clock rate over cycle count. However, we expect that the TRIPS microarchitecture, with its partitioned design and no global wires, could be implemented in a clock rate equivalent to the Core 2, given a custom design and the same process technology. Another pitfall is that the relatively slow clock rate of TRIPS may make memory accesses less expensive relative to high clock-rate processors. To counter this effect, we under-clock the Core 2 from 1.8 GHz to 1.6 GHz to make the processor/memory speed more similar to that of TRIPS. Because the benchmarks are largely L2 cache resident, the relative memory speed has little effect on application execution time.

Benchmarks: Table 2 shows our benchmarks, ranging from simple kernels to complex uniprocessor workloads, compiled with the TRIPS C and Fortran compiler [22]. The suite labeled *simple* refers to applications

Suite	Count	Benchmarks
Kernels	4	transpose (ct), convolution (conv), vector-add (vadd), matrix multiply (matrix)
VersaBench	3 of 10	bit and stream (fmradio, 802.11a, 8b10b)
EEMBC	28 of 30	Embedded benchmarks
Simple	15	Hand-optimized versions of Kernels, VersaBench, and 8 EEMBC benchmarks
SPEC 2K Int	9 of 12	All but gap, vortex and C++ benchmarks[1]
SPEC 2K FP	9 of 14	All but sixtrack and 4 Fortran 90 benchmarks[1]

Table 2: Benchmark suites.

with hand-optimizations: 4 application kernels, 3 stream and bit operation benchmarks from the VersaBench suite [17], and 8 medium-sized benchmarks from the EEMBC benchmarks [5]. We hand-optimized benchmarks to guide compiler efforts and explore the potential of the system. We performed hand-optimization on the compiler-generated TIL code and scheduled the result with the compiler. Most of the hand-optimizations are mechanical, but not yet implemented in the compiler. We more extensively hand-optimized four scientific kernels on TRIPS: matrix transpose (*ct*), convolution (*conv*), vector add (*vadd*), and matrix multiply (*matrix*); further, we hand-scheduled *matrix* and *vadd*.

The most complex benchmarks come from SPEC2000 and include 10 integer and 8 floating-point benchmarks [24]. Three SPEC programs that currently fail to build correctly with our toolchain are omitted. We use a consistent set of compiler flags for all benchmarks rather than tuning the flags for performance on a per-benchmark basis. We use SimPoint simulation regions for our simulation-based evaluation of the SPEC benchmarks [21].

4 ISA Evaluation

This section uses simulation to examine how well programs map to the TRIPS ISA, characterizing block size, instruction overheads, and code size. We compare TRIPS and RISC ISA (Alpha) statistics to quantify the relative overheads of the TRIPS ISA. We present details for the simple benchmarks and means for EEMBC, SPEC INT, and SPEC FP.

4.1 TRIPS Block Size and Composition

A key parameter for a block-atomic EDGE ISA is the block size. Early experience demonstrated that creating programs with average block sizes of 20+ instructions was not difficult with standard compiler transformation and that larger blocks would increase the instruction window, better amortize block overheads, and have the potential for better performance. Seeking this performance, we chose to push the compiler technology by selecting 128-instruction block sizes.

[1]Section 5 omits *ammp* and *parser* as they do not execute correctly on the TRIPS microarchitecture simulator.

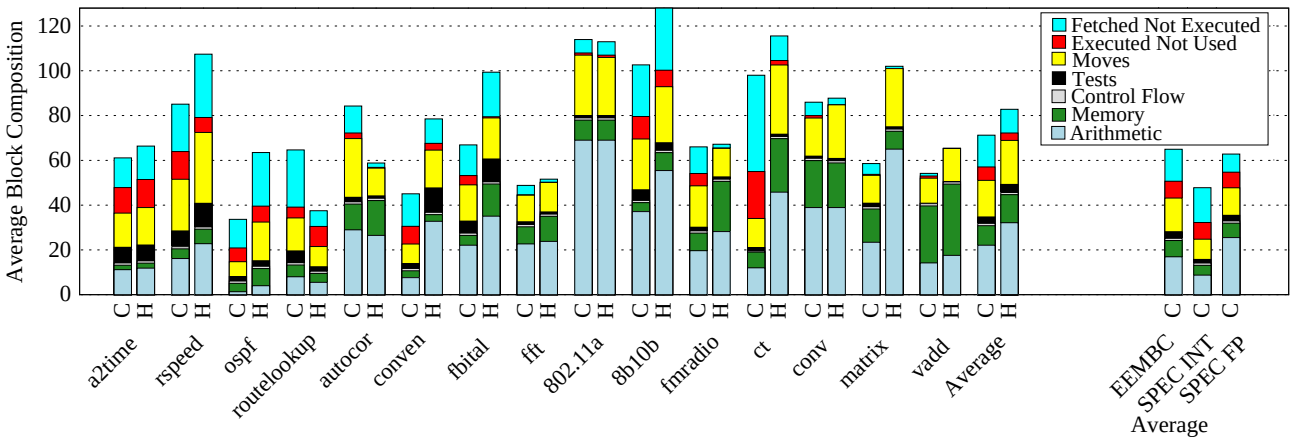

Figure 3: TRIPS block size and composition for compiled (C) and hand-optimized (H) benchmarks.

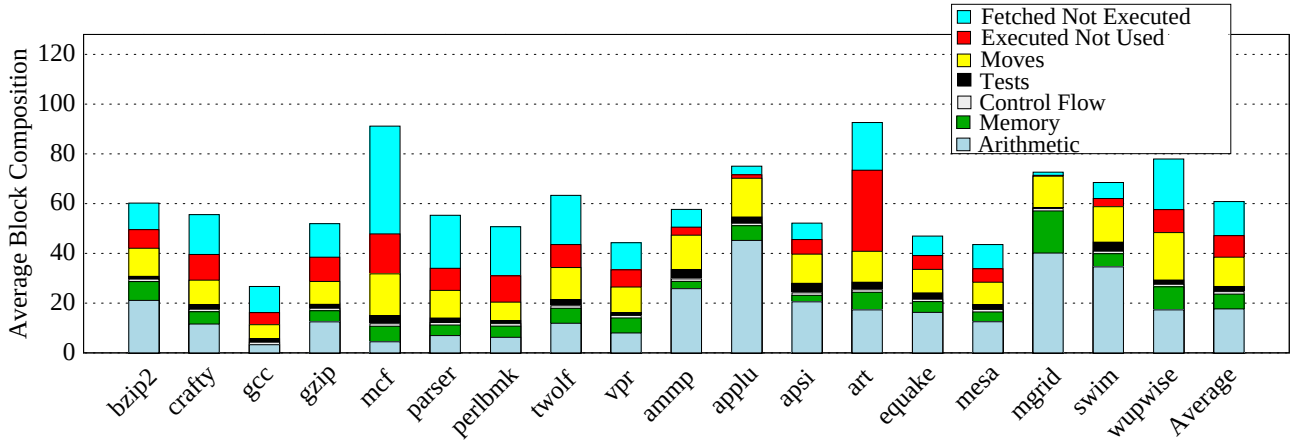

Figure 4: TRIPS block size and composition for compiled SPEC benchmarks.

Figure 3 shows the average block size weighted by execution frequency and broken down by the number of arithmetic instructions, memory instructions, branch/jump/call/return instructions, test instructions (used for branches and predication), and `move` instructions (used to fan out intermediate operands). The figure does not include the register read/write instructions, which reside in the block header and not in the 128 instructions. *Fetched Not Executed* instructions in a block are never executed either because they did not receive a matching predicate or because they did not receive all of their operands due to predicated instructions earlier in the block's dataflow graph. *Executed Not Used* instructions were fetched and executed speculatively but their values were unused due to predication later in the dependence graph.

For some programs, such as *a2time*, mispredicated instructions account for a third of the instructions within a block. *A2time* contains several nested `if/then/else` statements. To fill blocks and minimize executed blocks, the compiler produces code that speculatively executes both `then` and `else` clauses simultaneously within one block and predicates to select the correct outputs. Aggressive predication can improve system

performance by eliminating branch mispredictions and increasing front-end fetch bandwidth.

The remainder of the instruction types, tests, control flow, memory, and arithmetic, are required for correct execution. The number of useful instructions (excluding `move` and mispredicated instructions) varies. Some programs with complex control have only 10 instructions per block while others with more regular control have as many as 80 instructions per block. To implement dataflow execution in a block, the TRIPS ISA uses `move` instructions. Because TRIPS instructions have fixed width (32 bits), arithmetic and load instructions can target at most two consumers. The compiler therefore inserts `move` and special `move3` and `move4` instructions to fanout values consumed by more than two instructions. Predicate merge points may require predicated `move` instructions. The result is that `move` instructions account for nearly 20% of all instructions, more than anticipated at the start of the design. Support for wider fanout move instructions (multicast) would substantially reduce this overhead.

Compiled code has an average block utilization of 70 instructions, but with high variance, ranging from 35 to over 110 instructions. Hand-optimizations execute fewer blocks by increasing block utilization. For example, the hand-optimized version of *ospf* has blocks two times larger than its compiled versions. Hand-optimizations include eliminating unnecessary instructions and then merging adjacent smaller blocks or increasing unrolling factors to fill blocks. Higher block utilization is correlated with higher performance. *Routelookup* and *autocor* actually have smaller block size when hand-optimized but a similar number of useful instructions. These programs are memory and control bound; larger blocks do not improve performance due to the predication overhead. Both hand-optimized and compiled code utilize the aggressive 128-instruction block size to achieve average block sizes ranging from 20 to 128.

Figure 4 shows the block composition for each of the SPEC benchmarks. We see that on average the integer benchmarks both have smaller blocks and rely more heavily on predication than the floating point benchmarks. This is a due to the control intensive nature of the integer benchmarks which makes forming large hyperblocks a challenge.

4.2 TRIPS ISA versus Alpha

To quantify the differences between the TRIPS ISA and a RISC ISA, we compare to the Alpha. Figure 5 shows fetched instruction counts on TRIPS normalized to Alpha, with TRIPS including neither register read/write instructions from the block header nor `NOP`s in underfull blocks. For both TRIPS and Alpha, the instruction count omits incorrectly fetched instructions due to branch mispredictions.

The number of useful instructions varies widely by benchmark suite which is a function of the state of the TRIPS compiler and the gcc Alpha compiler. Figure 6 compares instructions between TRIPS and Alpha for the SEPC benchmarks. Overall, TRIPS executes half as many useful instructions on the simple benchmarks,

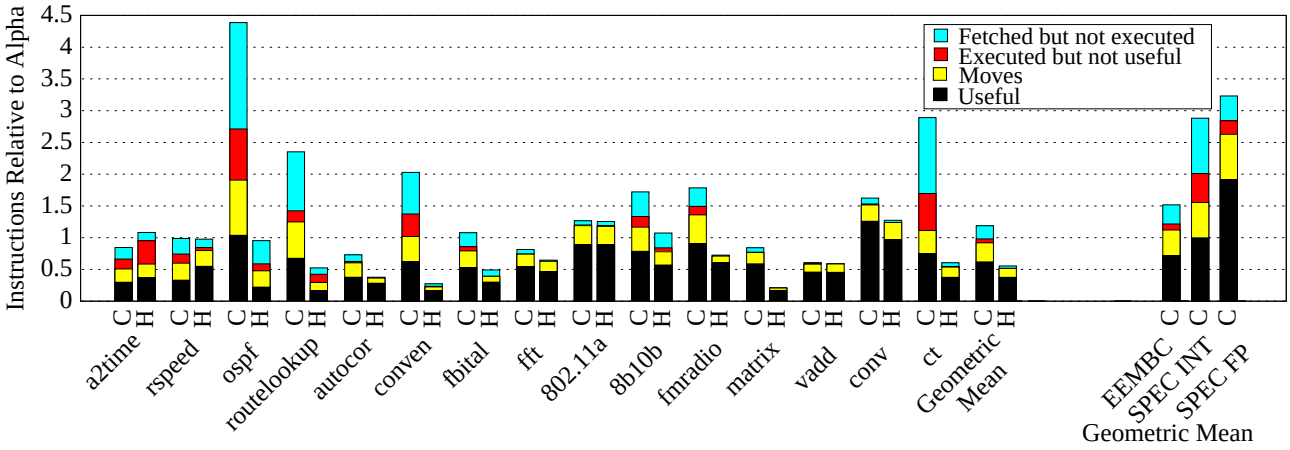

Figure 5: TRIPS instructions normalized to Alpha for compiled (C) and hand-optimized (H) benchmarks.

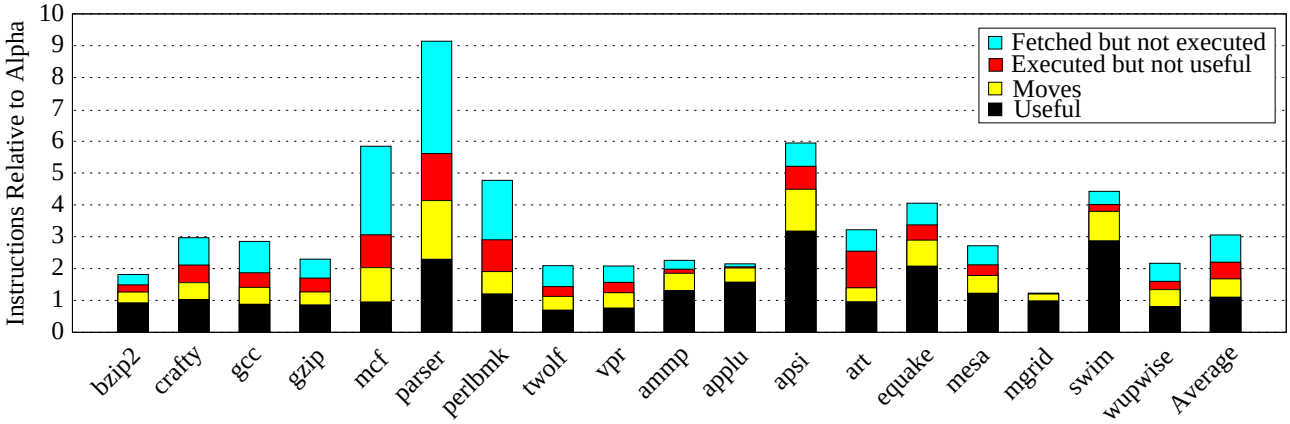

Figure 6: TRIPS instructions normalized to Alpha for compiled SPEC benchmarks.

an equal number on SPEC INT, and twice as many on SPEC FP. On compiled code, TRIPS tends to execute more instructions due to prototype simplifications, which introduce inefficiencies in constant generation and sign extension unrelated to its execution model. For hand-optimized benchmarks, TRIPS executes fewer instructions because its larger register set (128 registers) eliminates store/load pairs and because more aggressive unrolling exposes more opportunities for instruction reduction. The number of fetched but mispredicated instructions varies across the benchmarks, depending on the degree of predication. Overall, TRIPS may need to fetch as many as 2–4 times more instructions than the Alpha on the simple benchmarks and 2–9 times more instructions on the SPEC benchmarks, due to aggressive predication.

4.3 Register and Memory Access

TRIPS inter-block communication uses registers and memory while intra-block communication is direct between instructions, reducing the number of accesses to registers and memory. TRIPS has a total of 128

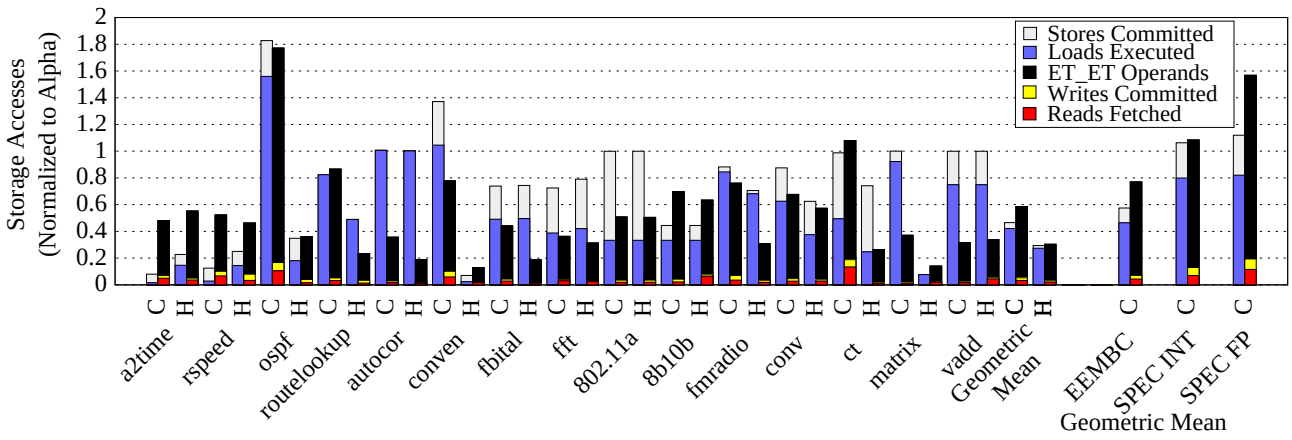

Figure 7: Storage accesses normalized to Alpha for compiled (C) and hand-optimized (H) benchmarks.

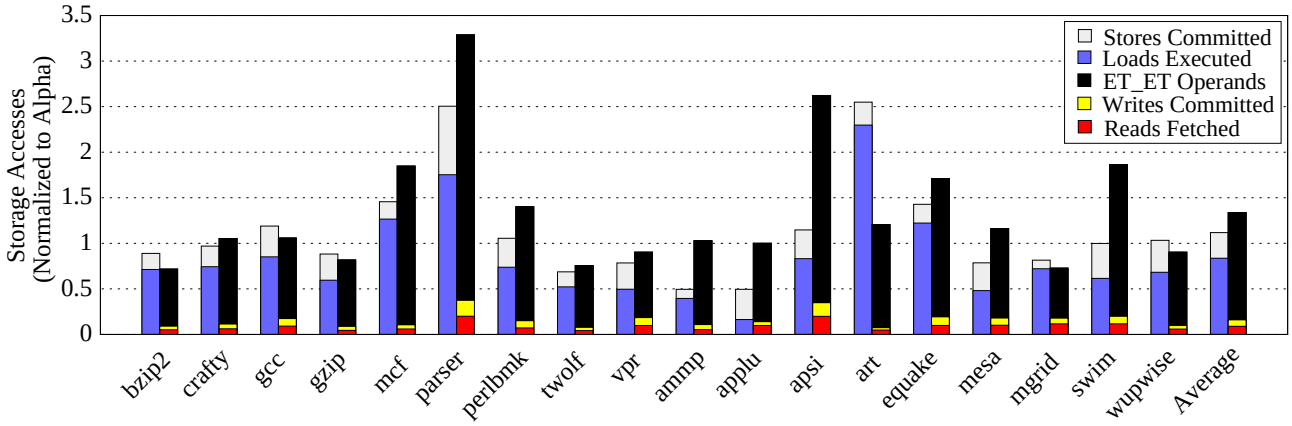

Figure 8: Storage accesses normalized to Alpha for compiled SPEC benchmarks.

registers spanning four register banks (32 registers per bank). Each bank has one read and one write port. The large register file reduces load on the memory system since the compiler can register allocate more of a program's variables [14]. Compared to a conventional architecture, TRIPS replaces memory instructions with less expensive register reads and writes, and replaces register reads and writes with less expensive direct communication between producing and consuming instructions.

The left bar stack of each pair in Figure 7 shows the number of loads and stores on TRIPS normalized to loads and stores on the Alpha. TRIPS executes about half as many memory instructions as the Alpha and as few as 10%, due to its larger register file and direct instruction communication. Several hand-optimized benchmarks have significantly fewer memory accesses than the compiled versions because they register allocate fields in structures and small arrays, whereas the compiler currently does not. The right bar stack shows the number of register reads, writes, and operand network communications on TRIPS normalized to register reads and writes on the Alpha. Because of direct operand communication, TRIPS accesses the register file

80–90% less often than the Alpha. The top bar shows direct operand communication that replaces register accesses on TRIPS.

Compared to their compiled counterparts, hand-optimized benchmarks generally have fewer register accesses, OPN communications, and memory accesses. The hand-optimized versions aggressively register allocate more memory accesses by using programmer knowledge about pointer aliasing, much of which may be automated. They also eliminate instructions, such as unnecessary sign extensions, which could be automated with aggressive peephole optimizations. On average, the sum of register reads, writes, and direct communications approximates the number of Alpha register reads and writes. Figure 8 shows that on some benchmarks (SPEC INT), direct communication is large because of the distribution of predicates and communication of useless values by mispredicated instructions. On SPEC FP the large number of accesses is a result of TRIPS executing many more instructions than Alpha. In a conventional architecture, the register file broadcasts an instruction's result to other instructions. In TRIPS, fanout may require a tree of `move` instructions, which increases communication and the number of instructions.

4.4 Code Size

The TRIPS ISA significantly increases dynamic code size over Alpha. Each block has 128 32-bit instructions, a 128-bit header, 32 22-bit read instructions, and 32 six-bit write instructions. The compiler inserts `NOP`s when a block has fewer than 32 reads/writes or fewer than 128 instructions. `NOP`s consume space in the L1 I-cache but are not executed. We compared dynamic code size of TRIPS to Alpha by counting the number of unique instructions that are fetched. Figures 9 and 10 show the incrase in dynamic code size across all of the benchmarks. On average the dynamic code size of TRIPS, including the overheads from the block header, read and write instructions, and nops, averages about 11 times larger than the Alpha, but with a wide variance. In general we see the largest increase in the memory footprint on the smaller benchmarks especially *matrix* and *vadd*. This is due to the aggresive optimizations done on the TRIPS platform that improve performance at the expense of code size. For example, on *vadd* the Alpha code is a single tight loop with just a few instructions in the loop body, while on the TRIPS code the loop was unrolled 64 times to simplify address calculation and reduce the network congestion experienced by cache accesses. Without the block header, read and write instructions, and the nop overheads, the number of unique instructions for TRIPS is 5 times that of Alpha, while the number of unique useful instructions for TRIPS (discounting the instructions that are fetched but not needed) is 2–3 times greater than Alpha. Thus instruction replication due to TRIPS block optimizations accounts for about half of the code bloat.

Experiments generally show a low instruction cache miss rate on small and medium sized benchmarks, but some SPEC benchmarks have miss rates in the range of 20–40%, indicating that cache pressure is a problem

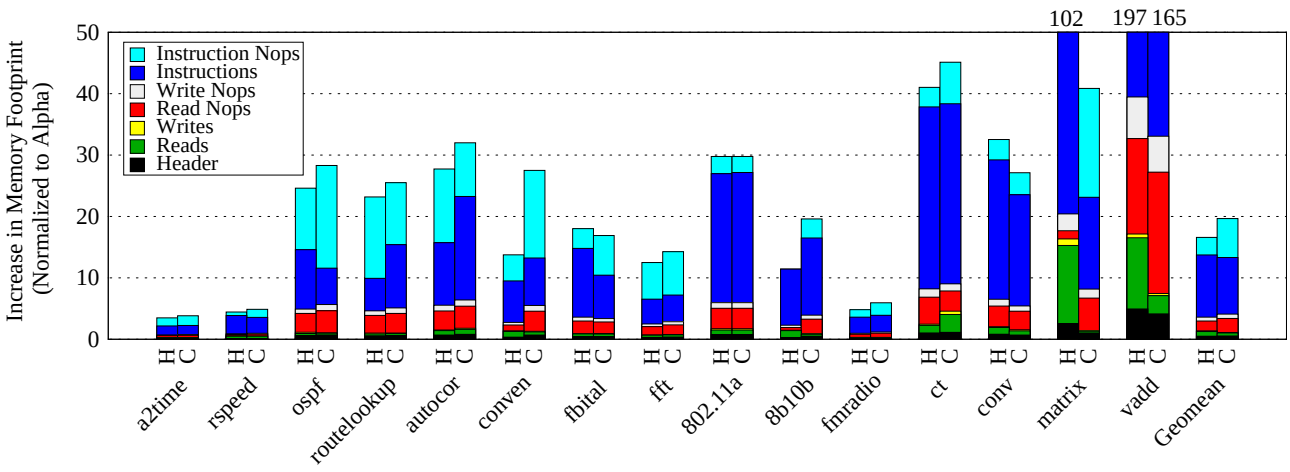

Figure 9: Increase in dynamic memory footprint for compiled (C) and hand-optimized (H) benchmarks.

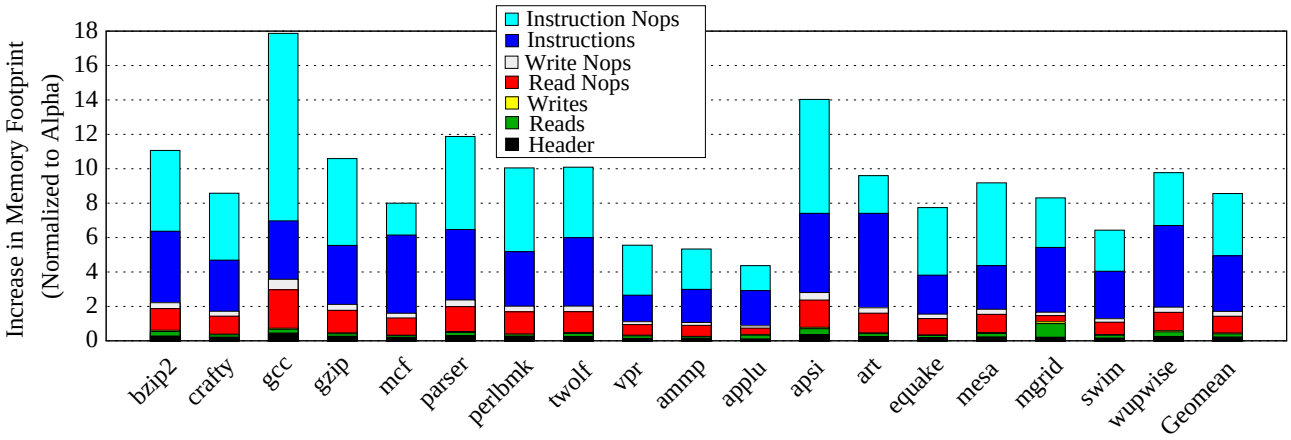

Figure 10: Increase in dynamic memory footprint for compiled SPEC benchmarks.

for some real applications. The TRIPS prototype can compress underfull instruction blocks in memory and in the L2 cache down to 32, 64, or 96 instructions, depending on block capacity, which reduces the expansion factor over Alpha from 11 to 6. Block compression in the instruction cache may unduly slow down instruction fetch or require more complex instruction routing from the instruction cache banks to the execution tiles. The results indicate that the benefits of variable block sizes warrant this complexity for future designs. Furthermore, increasing the instruction cache size in distributed architectures is relatively easy and will also mitigate cache pressure.

5 Microarchitecture Evaluation

5.1 Filling a 1K Instruction Window

With up to 128 instructions per block and eight concurrently executing blocks, TRIPS has a maximum dynamic instruction window size of 1024 instructions. Figure 11 shows the average number of TRIPS in-

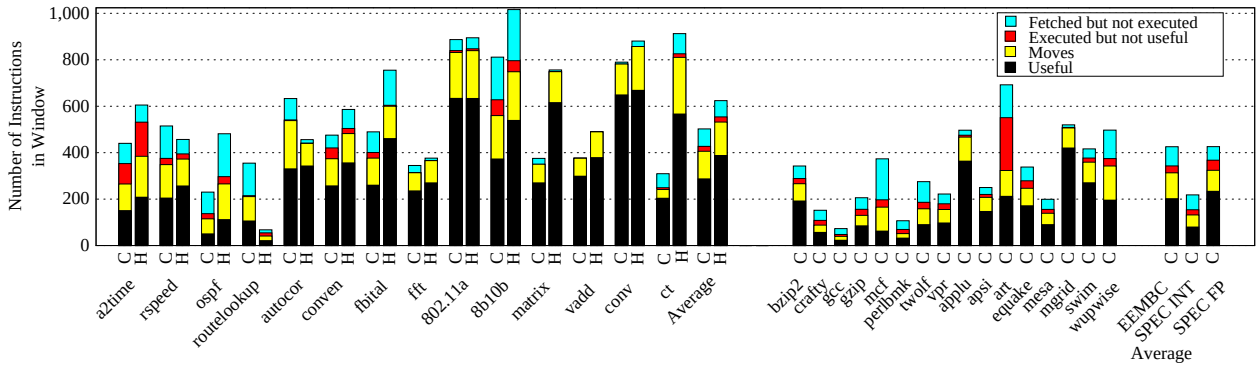

Figure 11: Average number of in-flight instructions for compiled (C) and hand-optimized (H) benchmarks.

structions in the window. This metric multiplies the average number of blocks in flight (speculative and non-speculative) and the average number of instructions per block. Compiled codes have on average 400 total instructions of which more than 170 are useful. The hand-optimized programs with larger blocks achieve a mean of 630 instructions, more than 380 of which are useful. Compared with issue windows of 64 to 80 on modern superscalar processors, TRIPS exposes more concurrency at the cost of more communication due to the distributed window.

The principal speculation mechanisms in TRIPS are predication, load-store dependence prediction, and next-block prediction. When the load/store queue detects that a speculative load is incorrect, it flushes the block pipeline and enters the load into the dependence predictor's partitioned load-wait table. The predictor is effective in part because the compiler reduces the number of loads and stores (as discussed in Section 4.3). For the SPEC benchmarks, TRIPS flushes fewer than one block per 2000 useful instructions, without overly constraining speculative load issue.

The TRIPS next-block predictor selects the next speculative block [19]. It consists of a 5 KB local/global tournament exit predictor that predicts the exit branch (one of up to eight) from the block and a 5 KB multi-component target predictor that predicts the target address of this exit. Figure 12 shows the prediction breakdown for four different configurations: (*A*) shows an Alpha 21264-like conventional tournament branch predictor (10 KB) predicting TRIPS-compiled *basic blocks*; (*B*) shows the TRIPS block predictor (10 KB) predicting basic blocks; (*H*) shows the TRIPS block predictor (10 KB) predicting optimized TRIPS blocks, and (*I*) shows a "lessons learned" block predictor (14 KB) that scales up the target predictor component sizes to 9 KB. Each bar is normalized to the total number of predictions made for basic blocks to measure accuracy and reductions in predictions due to TRIPS blocks. The average MPKI (Mispredictions Per 1000 Instructions, omitting move and mispredicated instructions) observed for these four configurations on SPEC INT are 14.9, 15.1, 8.6 and 7.3 respectively. SPEC FP applications have an MPKI of 1.6, 1.7, 1.5 and 1.3 respectively.

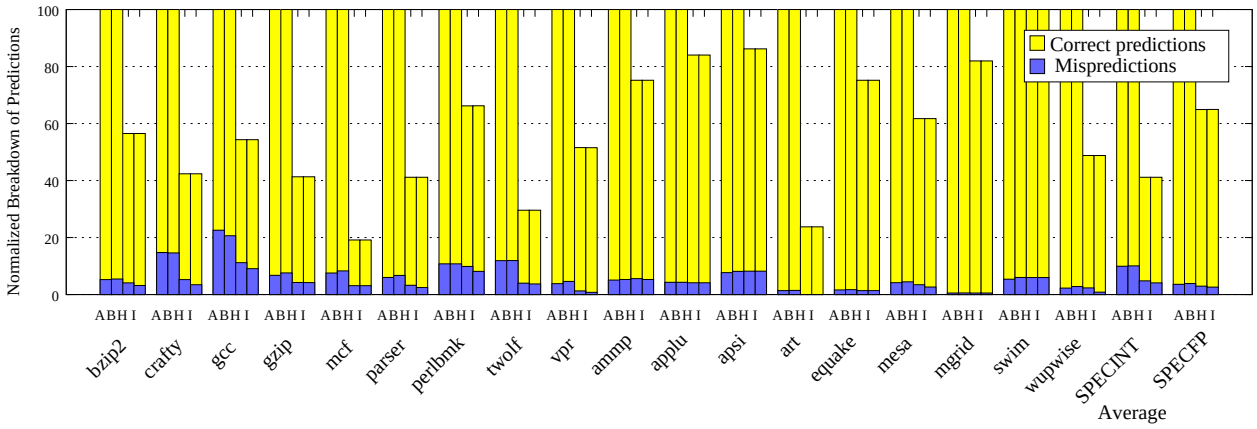

Figure 12: Block-predictor mispredictions normalized to total predictions made for basic blocks.

	L1 D-Cache to Processor	L2 to L1	Memory to L2
Peak Ops/ cycle	4 - 8byte requests	3.2 - 16 byte requests	1 - 64 byte request
Peak BW (GBytes/sec)	10.9	17.5	5.6
Achieved BW (GBytes/sec)	10.5	17.2	3.2
% of Peak	96.5%	98.5%	57.8%

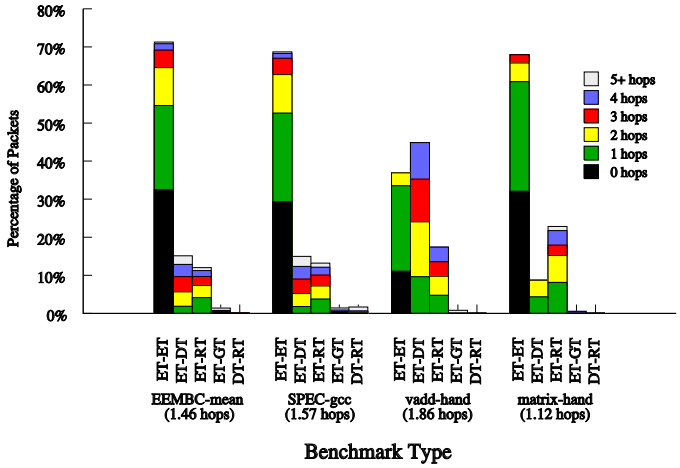

Figure 13: TRIPS bandwidths at 366MHz and operand network (OPN) profile with average hops per packet

The accuracy of predicting predicated blocks is neither strictly better nor strictly worse than that of predicting basic blocks. Predication within TRIPS blocks may improve accuracy by removing hard-to-predict branches, but may also degrade accuracy by obscuring correlating branches in the history. Although the TRIPS predictor (*H*) has a higher misprediction rate (18% higher) than a conventional predictor (*A*), it has a lower MPKI because it makes fewer predictions—59% fewer on SPEC INT and 35% fewer on SPEC FP. The improved TRIPS predictor (*I*) reduces SPEC INT MPKI by 15.8% and SPEC FP MPKI by 14.2%. Lower prediction accuracy has a significant effect on the instruction window utilization and has a strong correlation with performance. However, more aggressive next-block predictors may still fall short of modern branch prediction accuracies. Increasing the size of the branch target buffer, call target buffer, and history register does improve accuracy. Advanced multi-component long-history predictors [9, 20] will likely also improve exit and target accuracy and consequently performance.

5.2 Feeds and Speeds

In this section, we explore the performance of the banked memory system and operand network, two defining features of the distributed TRIPS implementation. For the memory system, we developed kernels that saturate the bandwidth of each of the banks, providing insight into the types of optimizations required for memory-bound programs. For the operand network, we measure traffic load to determine how well the compiler's placement algorithm minimizes the distance between communicating instructions.

Memory System: The TRIPS prototype employs an address-partitioned memory system that divides the L1 data cache into four 8-KB, single-ported data banks and the L2 cache into sixteen 64-KB, single-ported memory banks. The table in Figure 13 shows the achieved memory bandwidth on the hardware at a core speed of 366 MHz for a hand-optimized vector add (`vadd`) kernel. With careful instruction placement, *vadd* can attain nearly 100% of the core's peak of four memory operations per cycle (10.5 GB/sec), indicating effective use of the partitioned L1 data cache. By adjusting the vector size of *vadd*, we constructed a microbenchmark with an access pattern to maximize the consumption of the L2 cache and main memory bandwidth. This program nearly reached the theoretical peak of the L2 and a majority of the main memory bandwidth provided by the dual DDR memory controllers. While the benchmark achieves only 57.8% of the maximum interface bandwidth, the vast majority of the loss is due to the memory controller protocol and not to the TRIPS design itself. Similar techniques and principles were used to hand-optimize dense matrix kernels [4] and lessons learned from these case studies were used to improve the compiler's instruction placement algorithms.

Operand Network: The Operand Network (OPN) connects the TRIPS processor tiles and transmits operands between execution tiles (ETs), the register file (RTs), and the data cache (DTs) [7]. The TRIPS scheduler optimizes instruction placement on the tile topology to exploit concurrency and minimize the distance between dependent instructions along the program's critical path. The graph in Figure 13 displays the breakdown of the hop count for OPN traffic. On average, about half of operands are bypassed locally within an ET. Of the traffic that must traverse the OPN, about 80% require one or two hops, resulting in an overall operand hop count of 0.9. Ideally, all operand communication would be bypassed locally (0 hops), but the inherent tradeoff between locality and concurrency combined with limited instruction storage per tile demands that many communicating instructions reside on different tiles. About 60% of the OPN messages stems from ET–ET operand traffic, with the remaining messages about evenly split between ET–DT and ET–RT traffic. On the SPEC benchmarks two-thirds of the ET–DT traffic and one half of the ET–RT traffic requires three or more hops because the DTs and RTs lie along the edge of the ET array. Simulations results show that congestion contributes only a 12% performance overhead as the latency due to hops count is more significant. These results indicate opportunities for on-chip network design innovations to improve performance of distributed

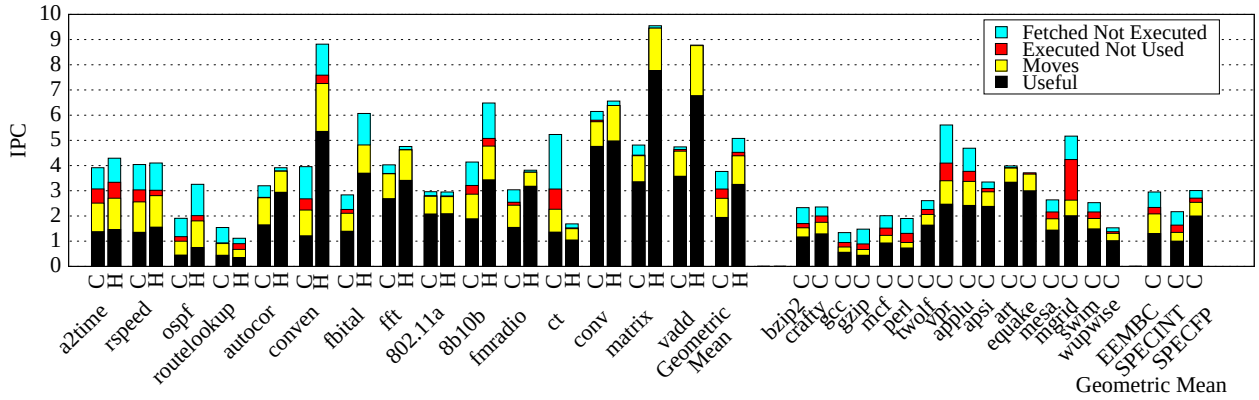

Figure 14: Instructions per clock (IPC) on compiled (C) and hand-optimized (H) benchmarks.

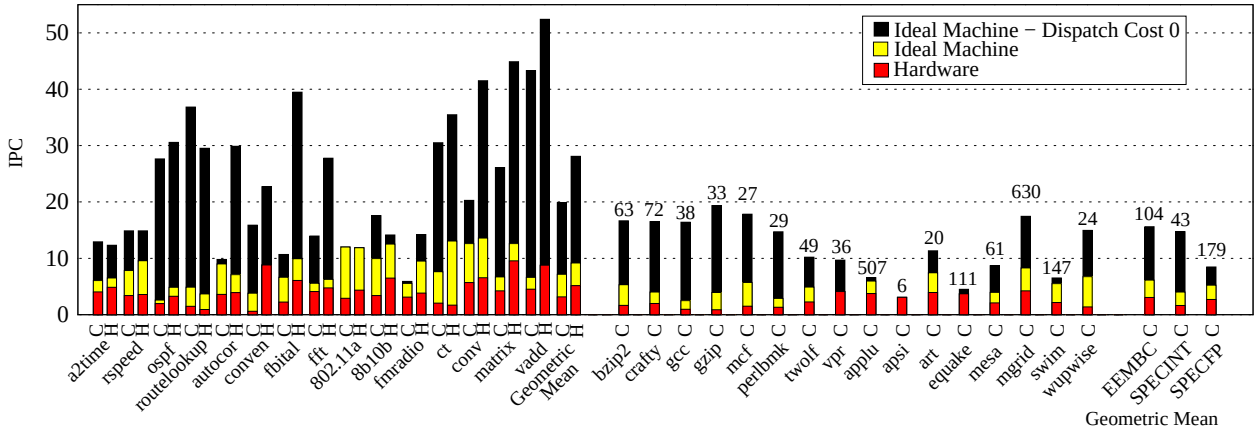

Figure 15: IPC for TRIPS and benefits for two idealized EDGE designs.

architectures.

5.3 ILP Evaluation

TRIPS executes up to 16 instructions per cycle, but can only sustain that rate under ideal conditions: 8 blocks full of executed instructions, perfect next-block prediction, and no instruction stalls due to long-latency instructions. Actual IPC is limited to 1/8 of the block size because of block fetch latency. Since the average block size of our hand-optimized benchmarks is 80 instructions, we could achieve at most an average IPC of 10 on them. Figure 14 shows the sustained IPC that TRIPS achieves across the benchmarks. While some applications are intrinsically serial (e.g., *routelookup* traverses a tree data structure serially for an IPC near 1), others reach 6 to 10 IPC, showing that the processor exploits more ILP in these programs. The hand optimized codes have an IPC 25% greater on average than their compiled counterparts, mostly due to executing fewer more densely packed blocks. The SPEC benchmarks have lower IPCs, both because they have smaller

average block sizes and more flushes due to branch mispredictions and i-cache misses.

To understand the theoretical ILP capability of EDGE architectures, we conducted a limit study using an idealized EDGE machine with perfect prediction, perfect predication, perfect caches, infinite execution resources, and a zero-cycle delay between tiles. Like TRIPS, we use a 1K window size and limit dispatch and fetch to one new block every eight cycles. Figure 15 shows that on average this ideal machine only outperforms the prototype by roughly a factor of 2.5, indicating only moderate room for improvement due to low inherent application ILP, dispatch cost, and limited window size. Simulating this ideal machine with a zero-cycle dispatch cost increases the IPC on average by a factor of four. However, eliminating only the dispatch delay on TRIPS improves performance by only 10%, which indicates that dispatch is not the primary bottleneck on the hardware. We annotate the top of the SPEC bars with the IPC for the ideal machine with a 128K instruction window and a dispatch cost of zero cycles. The SPEC benchmarks have a wide range of available ILP, with most benchmarks around 50 IPC but some FP benchmarks having IPCs in the hundreds. The simple benchmarks have a similar range of IPCs. Several, such as *802.11a* and *8b10b*, are inherently serial and do not exceed 15. Others, such as *vadd* and *fmradio*, are quite concurrent with IPCs of 1000 and 500 respectively on the ideal machine with a 128K window, but are resource limited on the hardware. This study reveals that the amount of ILP currently available to TRIPS is limited and that larger window machines have the potential to further exploit ILP.

6 TRIPS versus Commercial Platforms

This section compares TRIPS to conventional processors using cycle counts from performance counters, which normalizes for different clock rates. We use hand-optimized benchmarks for TRIPS to show the potential of the system and compiled benchmarks to show the current state of the TRIPS compiler. We compare to the GNU C compiler (gcc) and the native Intel compiler (icc) on the reference machines to identify the effect of platform-specific optimizations. The quality of scalar optimization in gcc is more similar to the TRIPS compiler, since the TRIPS compiler is an academic research compiler. Consequently, we normalized performance to the Core 2 using the gcc compiler.

Simple Benchmarks: Figure 16 shows relative performance (computed as a ratio of cycles executed relative to the Core 2 using gcc) for TRIPS hand-optimized code, TRIPS compiled code, icc-compiled code for the Intel Core 2, and gcc-compiled code for the Intel Core 2, Pentium 4, and Pentium III. The TRIPS compiler achieves equivalent performance to the Core 2 on average, with better performance on nine benchmarks and worse performance on six. Benchmarks with smaller speedups (*rspeed*) employ sequential algorithms that do not benefit from increased execution bandwidth or deep speculation. The benchmarks that show the largest speedups (*matrix* and *8b10b*) typically have substantial parallelism exposed by the large window on TRIPS.

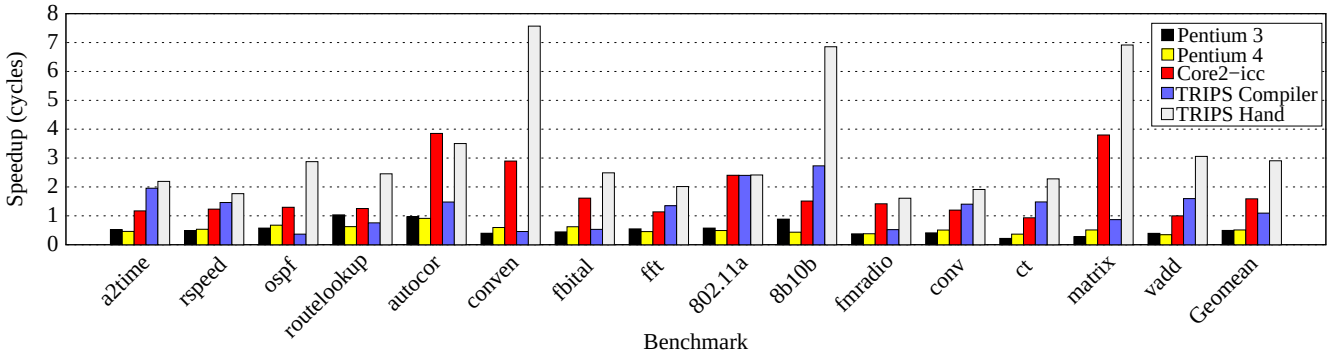

Figure 16: Speedup of TRIPS relative to the Core 2 on simple benchmarks.

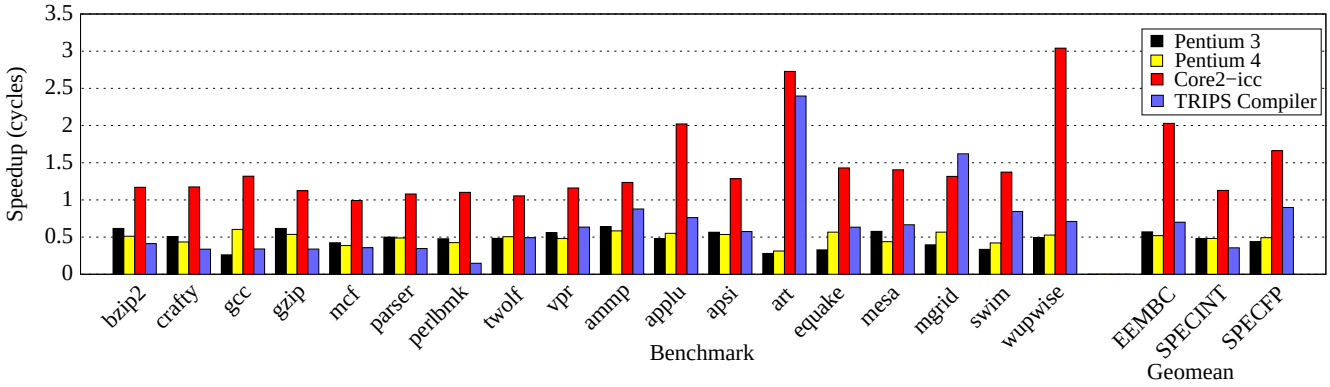

Figure 17: Speedup of TRIPS relative to the Core 2 on SPEC benchmarks.

The TRIPS hand-optimized code always outperforms the Core 2, with an average 2.9x cycle count reduction.

The performance differences between TRIPS compiled and hand-optimized code are primarily due to more aggressive block formation, unrolling, and scalar replacement. For example, *8b10b* benefits from unrolling the innermost loop of the kernel to create a full 128-instruction block and from register allocating a small lookup table. In *fmradio*, the hand-optimized code fuses loops that operate on the same vector, and uses profile information to exclude infrequently taken paths through the kernel.

To show the ability of the TRIPS architecture to exploit a large number of functional units, we compare a TRIPS hand-optimized and hand-scheduled matrix multiply [4] to the state-of-the-art hand-optimized assembly versions of GotoBLAS Streaming Matrix Multiply Libraries on Intel platforms [6]. We use the best published results from library implementations for conventional platforms (*not* the results in Figure 17). The performance across platforms, measured in terms of FLOPS Per Cycle (FPC), ranges from 1.87 FPC on the Pentium 4 to 3.58 FPC on the Core 2 using SSE. The TRIPS version achieves 5.20 FPC without the benefit of SSE, which is 40% better than the best Core 2 result.

SPEC CPU2000: Figure 17 compares performance on SPEC2000 using reference data sets. TRIPS per-

	Per 1000 useful instructions						
	Core 2 cond. br. misses	TRIPS cond. br. misses	TRIPS call/ret misses	Core 2 I-cache misses	TRIPS I-cache misses	TRIPS load flushes	Average useful insts in flight
bzip2	1.3	1.6	0.0	0.0	0.0	0.09	342.5
crafty	4.5	3.0	0.5	1.7	17.2	0.35	151.8
gcc	7.4	7.0	1.8	3.1	18.5	0.52	73.0
gzip	4.8	4.3	0.0	0.0	0.0	0.04	206.1
mcf	14.0	6.3	0.0	0.0	0.0	0.13	373.6
parser	2.0	3.2	0.1	0.0	0.6	0.04	—
perlbmk	2.5	0.4	8.3	0.0	13.0	0.19	106.9
twolf	8.5	4.8	0.1	0.0	8.2	0.36	275.2
vpr	0.5	1.4	0.5	0.0	3.2	0.40	221.8
ammp	0.2	1.5	0.1	0.0	1.0	0.05	—
applu	0.0	0.7	0.0	0.0	0.0	0.01	496.6
apsi	0.0	2.4	0.0	0.0	0.0	0.11	249.7
art	0.4	0.0	0.0	0.0	0.0	0.01	692.2
equake	0.2	0.6	0.0	0.0	0.9	0.08	337.9
mesa	1.4	1.6	0.0	0.0	3.5	0.04	199.4
mgrid	0.0	0.1	0.0	0.0	0.0	0.00	519.8
swim	0.0	1.0	0.0	0.0	0.0	0.00	416.1
wupwise	0.0	0.7	0.5	0.0	0.8	0.04	496.9

Table 3: Performance counter statistics for SPEC.

formance is much lower on the SPEC benchmarks than on the simple benchmarks. While floating-point performance is nearly on par with Core 2-gcc (Core 2-icc achieves a 1.9x speedup over TRIPS), integer performance is less than half that of the Core 2. Table 3 shows several events that have a significant effect on performance: conditional branch mispredictions, call-return mispredictions, I-cache misses, and load flushes for TRIPS, normalized to events per 1000 useful TRIPS instructions. Also shown are the branch mispredictions and I-cache misses for the Core 2, normalized to the same 1000 TRIPS instruction-baseline to ease cross-ISA comparison. The rightmost column shows the average useful TRIPS instructions in the window, from Figure 11.

Several of the SPECINT benchmarks have frequent I-cache misses, such as *crafty*, *gcc*, *perlbmk*, and *twolf*. These benchmarks are known to stress the instruction cache, and the block-based ISA exacerbates the miss rate because of TRIPS code expansion and the compiler's inability to fill the fixed-size 128-instruction blocks. *Perlbmk* also has an unusually high number of call/return mispredictions, due to an insufficiently tuned call and branch target buffer in TRIPS. All of these factors reduce the utilization of the instruction window; for example, *gcc* has an average of only 73 useful instructions in flight, out of a possible 121 based on the average block size. While the TRIPS call/return flushes and I-cache misses cause serious performance losses, branch mispredictions are competitive with the Core 2 and load dependence violations are infrequent. Benchmarks that have the most useful instructions in the window compare best to Core 2, such as *art* and *mgrid*. These benchmarks are known to contain parallelism, and show good performance with little compiler or microarchitectural tuning.

7 Lessons Learned

The prototyping effort's goals were twofold: to determine the viability of EDGE technology and to learn the right (and wrong) ways to build an EDGE machine. This design and evaluation effort taught us the following lessons about how to build this class of architectures.

EDGE ISA: Prototyping has demonstrated that EDGE ISAs can support large-window, out-of-order execution with less complexity than an equivalent superscalar processor. However, the TRIPS ISA had several significant weaknesses. Most serious was the limited fanout of the `move` instructions, which results in far too many overhead instructions for high-fanout operations. The ISA needs support for limited broadcasts of high-fanout operands. In addition, the binary overhead of the TRIPS ISA is too large. The 128-byte block header, with the read and write instructions, adds too much per-block overhead. Future EDGE ISAs should shrink the block header to no more than 32 bytes and support variable-sized blocks in the L1 I-cache to reduce the NOP bloat, despite the increase in hardware complexity.

Compilation: The TRIPS compiler can generate correct code with reasonable quality for the TRIPS ISA, despite the new burdens the ISA places on the compiler. We believe that an industrial production compiler could achieve code quality similar to our hand-optimized results because the most effective hand-optimizations are largely mechanical. Because of the challenges presented by block constraints, we moved structural optimizations, such as loop unrolling and hyperblock formation, to the back end after code generation. A remaining challenge is how best to form large blocks in control-intensive code. For example, frequent function calls that end blocks too early cannot be solved by inlining without substantial re-engineering to move this optimization from its traditional position in the front end to the back end. Another opportunity is to allocate more variables in registers, which requires better alias analysis of pointer data structures; the best hand-generated code replaced store-load pairs with intra-block temporary communications, producing tighter code and higher performance.

Microarchitecture: A microarchitecture with distributed protocols is feasible; the fully functional silicon indicates that the tiled nature of the architecture aided in both design and validation productivity. Another positive result is that the design eliminates distributed block control protocols (fetch, dispatch, commit, and flush) from the critical path. However, a number of artifacts in the microarchitecture resulted in significant performance losses. Most important was traffic on the operand network, which averaged just under one hop per operand. This communication resulted in both OPN contention and communication cycles on the critical path. Follow-on microarchitectures must map instructions, in coordination with the compiler, so that most instruction-to-instruction communication occurs on the same tile. The second most important lesson was that performance losses due to the evaluation of predicate arcs was occasionally high, since arcs that could

have been predicted as branches are deferred until execution. Future EDGE microarchitectures must support predicate prediction to evaluate the most predictable predicate arcs earlier in the pipeline. Third, the primary memory system must be distributed among all of the execution tiles; the cache and register bandwidth along one edge of the execution array was insufficient for many bandwidth-intensive codes.

8 Conclusions

At its inception, the TRIPS design and prototyping effort addressed three questions: (1) whether a distributed, tiled, EDGE-based processor was feasible, (2) whether EDGE ISAs form a manageable compiler target, and (3) whether an EDGE-based processor can support improved general-purpose, single-threaded performance. This evaluation shows that the TRIPS ISA and microarchitecture are in fact feasible to build, resulting in a tiled design that exploits out-of-order execution over a window of many hundreds of instructions. Despite the inter-tile routing latencies, the combination of the large window, dynamic issue, and highly concurrent memory system permits TRIPS to sustain up to 10 IPC, showing an average three-fold cycle count speedup over a Core 2 processor, if hand-optimized kernels are used.

However, the compiled cycle counts on major benchmarks, such as SPECINT and SPECFP, are not competitive with industrial designs, despite the greater computational resources present in TRIPS. On compiled SPEC2000 benchmarks, measuring cycle counts, the TRIPS prototype achieves 60% of the performance of a Core 2 running SPEC2000 compiled at full optimization with gcc. Despite the fact that the TRIPS design was built by fewer than twenty people in an academic environment, this level of performance does not support the hypothesis that EDGE processors could outperform leading industrial designs on large, complex applications. Even doubling the TRIPS performance would likely result in speedups too small to justify a switch to a new class of ISAs. These limitations are due partially to inefficiencies in the ISA and microarchitecture, but may also result from mismatches between certain program features and EDGE ISAs. For example, benchmarks with many indirect jumps, or unusually complex call graphs with many small functions, are difficult to compile into large blocks without a debilitating increase in binary size.

Nevertheless, the TRIPS prototype was a first-generation design, being compared to an extremely mature model, and there is much low-hanging fruit remaining in EDGE designs. The prototyping effort taught several lessons that result in significant improvements in both power and performance. Future EDGE designs should have support for variable-sized blocks, multicast of operands, predicate prediction, a more distributed/scalable memory system, smaller block headers, and less distributed mappings of instructions to tiles [18]. Also, since not all codes have high concurrency, future EDGE-based microarchitectures must allow adaptive granularity, providing efficient small configurations when larger configurations provide little performance benefit [11]. We project that these improvements will enable EDGE designs to outperform high-end commodity systems on

complex integer codes, but not by enough to justify deployment in full-power desktop systems. In the five-to-ten watt space, however, the performance and potential energy efficiency of EDGE designs may be sufficiently large to justify adoption in mobile systems or data centers, where high performance at low power is essential.

Acknowledgments

This research is supported by a Defense Advanced Research Projects Agency contract F33615-01-C-4106 and by NSF CISE Research Infrastructure grant EIA-0303609.

References

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. In *IEEE Micro*, pages 52–60, July/August 2006.

[2] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[3] K. Coons, X. Chen, S. Kushwaha, D. Burger, and K. McKinley. A Spatial Path Scheduling Algorithm for EDGE Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–140, October 2006.

[4] J. Diamond, B. Robatmili, S. W. Keckler, K. Goto, D. Burger, and R. van de Geijn. High Performance Dense Linear Algebra on Spatially Partitioned Processors. In *Symposium on Principles and Practice of Parallel Programming*, pages 63–72, February 2008.

[5] `http://www.eembc.org`.

[6] K. Goto and R. A. van de Geijn. Anatomy of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software*, 34(12):4–29, May 2008.

[7] P. Gratz, K. Sankaralingam, H. Hanson, P. Shivakumar, R. McDonald, S. W. Keckler, and D. Burger. Implementation and Evaluation of a Dynamically Routed Processor Operand Network. In *International Symposium on Networks-on-Chip*, pages 7–17, May 2007.

[8] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, July 2008.

[9] D. Jiménez. Piecewise Linear Branch Prediction. In *International Symposium on Computer Architecture*, pages 382–393, June 2005.

[10] C. Kim, D. Burger, and S. W. Keckler. An Adaptive Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[11] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, S. W. Keckler, and D. Burger. Composable Lightweight Processors. In *International Symposium on Microarchitecture*, pages 281–294, December 2007.

[12] B. Maher, A. Smith, D. Burger, and K. S. McKinley. Merging Head and Tail Duplication for Convergent Hyperblock Formation. In *International Symposium on Microarchitecture*, pages 65–76, December 2006.

[13] S. Melvin and Y. Patt. Enhancing Instruction Scheduling With a Block-Structured ISA. *International Journal on Parallel Processing*, 23(3):221–243, June 1995.

[14] A. Moshovos and G. S. Sohi. Speculative Memory Cloaking and Bypassing. *International Journal of Parallel Programming*, 27(6):427–456, December 1999.

[15] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *International Symposium on Microarchitecture*, pages 40–51, December 2001.

[16] PAPI: Performance Application Programming Interface. `http://icl.cs.utk.edu/papi`.

[17] R. M. Rabbah, I. Bratt, K. Asanovic, and A. Agarwal. Versatility and VersaBench: A New Metric and a Benchmark Suite for Flexible Architectures. Technical Report TM-646, Laboratory for Computer Science, Massachusetts Institute of Technology, June 2004.

[18] B. Robatmili, K. E. Coons, D. Burger, and K. S. McKinley. Strategies for Mapping Data Flow Blocks to Distributed Hardware. In *International Symposium on Microarchitecture*, pages 23–34, November 2008.

[19] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *International Symposium on Microarchitecture*, pages 480–491, December 2006.

[20] A. Seznec and P. Michaud. A Case for (Partially) TAgged GEometric History Length Branch Prediction. *Journal of Instruction-Level Parallelism*, Vol. 8, February 2006.

[21] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, September 2001.

[22] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. Burrill. Compiling for EDGE Architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, March 2006.

[23] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow Predication. In *International Symposium on Microarchitecture*, pages 89–102, December 2006.

[24] `http://www.spec.org`.

[25] B. Yoder, J. Burrill, R. McDonald, K. Bush, K. Coons, M. Gebhart, M. Govindan, B. Maher, R. Nagarajan, B. Robatmili, K. Sankaralingam, S. Sharif, A. Smith, D. Burger, S. W. Keckler, and K. S. McKinley. Software Infrastructure and Tools for the TRIPS Prototype. In *Workshop on Modeling, Benchmarking and Simulation*, June 2007.